USB-2200 Series

I/O Module User Manual

(Linux Version)

V1.0.0 March 2026

Written by Winson Chen

Warranty

All products manufactured by ICP DAS are under warranty regarding defective materials for a period of one year, beginning from the date of delivery to the original purchaser.

Warning

ICP DAS assumes no liability for any damage resulting from the use of this product. ICP DAS reserves the right to change this manual at any time without notice. The information furnished by ICP DAS is believed to be accurate and reliable. However, no responsibility is assumed by ICP DAS for its use, nor for any infringements of patents or other rights of third parties resulting from its use.

Copyright

Copy right © 2026 by ICP DAS Co., Ltd. All rights are reserved.

Trademarks

Names are used for identification purposes only and may be registered trademarks of their respective companies.

Contact Us

If you have any problems, please feel free to contact us.
You can count on us for a quick response.
Email: service@icpdas.com

Contents

1. Getting Started

This chapter guides users through hardware connection, driver installation and SDK/API deployment for the USB-2200 series, ensuring users can quickly start and develop applications.

After receiving the USB-2200 series, please verify that all components are included and undamaged. If any item is missing or defective, contact your supplier or distributor immediately for assistance.

USB I/O Module

USB Cable

Screw Driver

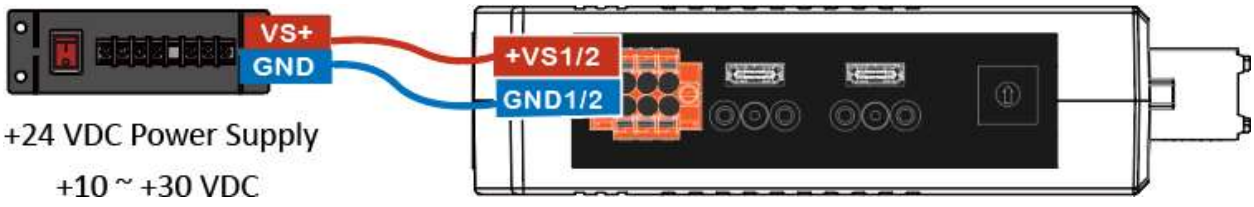
Quick Start Guide

1.1. Hardware Installation

This section explains how to connect the USB-2200 series to your PC for the first time. It includes the detection process of the USB device as a removable drive, and introduces how the Board ID is reflected in the device name.

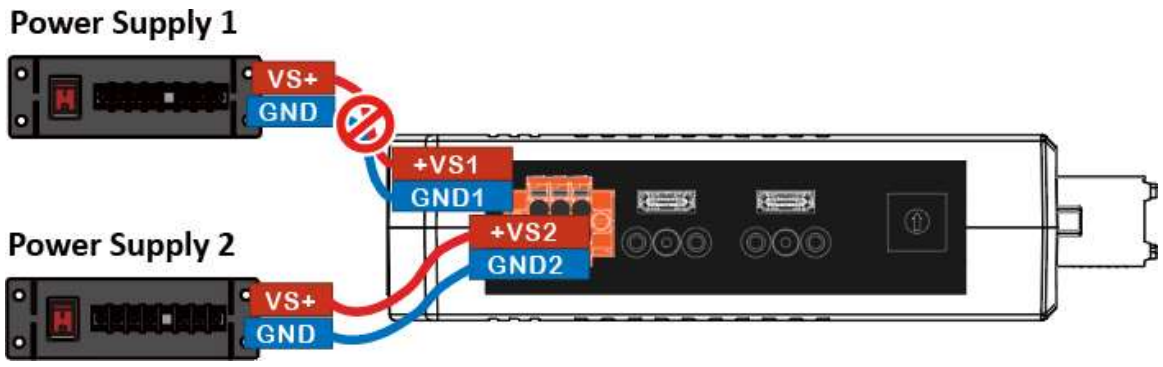
Step 1. Connect the power supply

- Connect the positive terminal (+) of the power supply to the terminal +VS1/2
- Connect the negative terminal (-) of the power supply to the GND1/2



Tips & Warnings

1. The USB-2200 series provides two independent power inputs that can be connected simultaneously.
2. If one power source fails, the other will automatically take over, ensuring redundant power backup and non-stop operation.

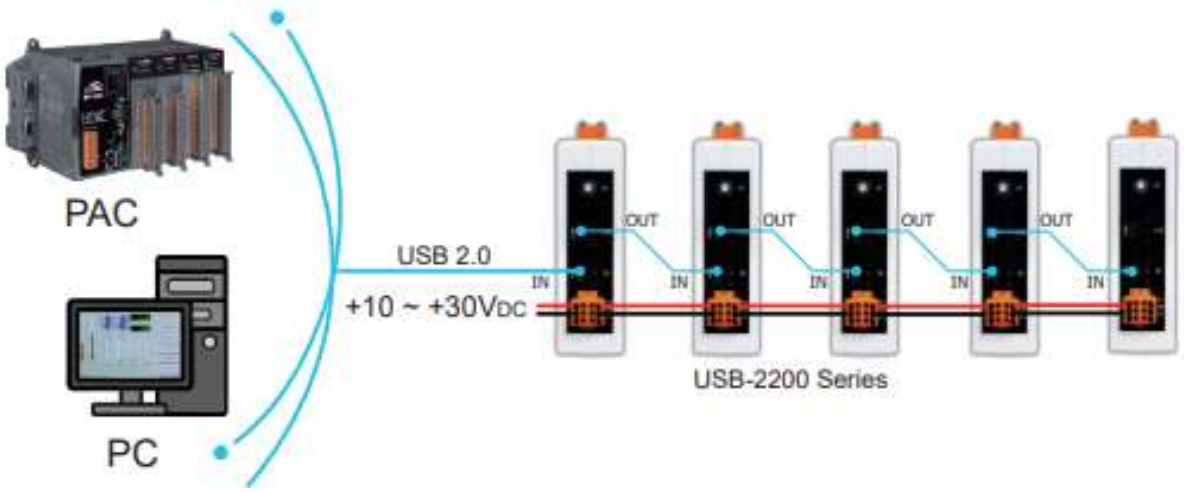


Step 2. Set the Board ID

- Configure the Board ID using the DIP switch or rotary switch (depending on the model).
- Each USB-2200 series in the system must have a unique Board ID when multiple modules are connected.

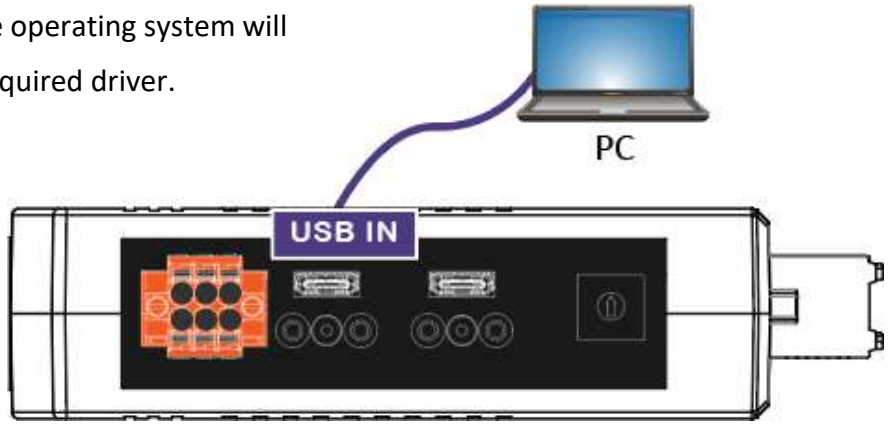
Tips & Warnings

- When two or more identical modules are connected, each must have a unique Board ID to avoid conflicts.
- The USB-2200 series supports Daisy-Chain connection, allowing multiple modules to be connected in sequence. When using Daisy-Chain, make sure that each module has a unique Board ID for correct identification and communication.



Step 3. Connect the USB Cable

- Use the provided USB Type-B to Type-A cable to connect the USB-2200 series to the host PC.
- Upon the first connection, the operating system will automatically install the required driver.



Step 4. Verify Power Status

- Verify that the Power LED on the USB-2200 series is lit.
- If the LED is not illuminated, recheck the power connections.

1.2. Software Installation

This section explains how to download, install, and run the USB I/O demo and related SDK for functional testing and software development. Users will learn how to obtain the required software package, install it properly, and begin using the USB I/O Utility for device testing and configuration.

Step 1. AccessDownload Center

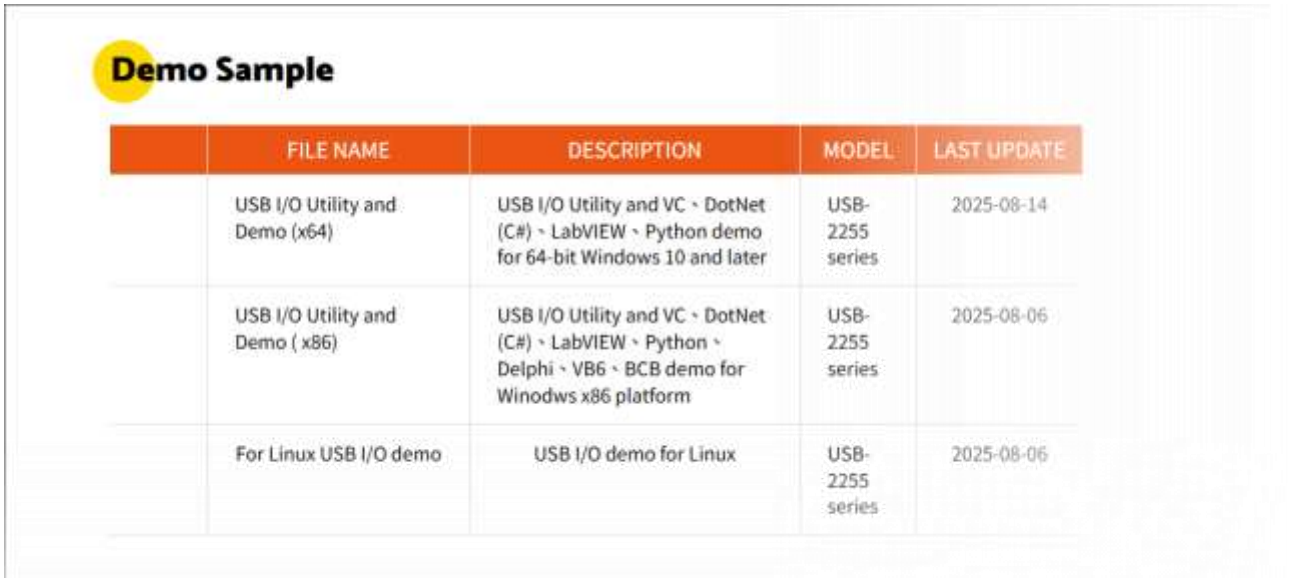
Go to the ICP DAS official **Download Center**:
https://www.icpdas.com/en/download/index.php?model=USB-2255%20series

Step 2. Select the SDK and utility package

In the Download Center, select the SDK and utility package according to your **operating system**. Choosing the correct version ensures compatibility and proper functionality.

File name

- **For Linux USB I/O demo**



Demo Sample

	FILE NAME	DESCRIPTION	MODEL	LAST UPDATE
	USB I/O Utility and Demo (x64)	USB I/O Utility and VC、DotNet (C#)、LabVIEW、Python demo for 64-bit Windows 10 and later	USB-2255 series	2025-08-14
	USB I/O Utility and Demo (x86)	USB I/O Utility and VC、DotNet (C#)、LabVIEW、Python、Delphi、VB6、BCB demo for Winodws x86 platform	USB-2255 series	2025-08-06
	For Linux USB I/O demo	USB I/O demo for Linux	USB-2255 series	2025-08-06

Step 3. Download and Extract the File

Download the selected package and extract the contents.

tar zxvf usb-2200.tgz
#cd usb-2200
#ls

```
examples  include  lib  Makefile
```

The software package includes:

- examples **– Example source codes demonstrating program usage.**
- Includes **– Header files for software development integration..**
- lib **– Provides two types of libraries for development:**

Static Library (.a): libusb2200.a – Use this to link the library functions directly into your application.

Dynamic Library (.so): libusb2200.so – These are dynamic link libraries required for device communication at runtime. Ensure the .so file is located in the system library path (e.g., /usr/lib) or defined in LD_LIBRARY_PATH.

Step 4. Verify USB-2200 device connection

cd usb-2200/examples/
#./list_device

```
Found 2 USB-2200 device(s):
 [0] PID=USB2255_64  BID=0x2
 [1] PID=USB2255_32  BID=0x0
```

This demo lists all currently available devices along with their **PID** and **BID**.

2. Operation & Deployment

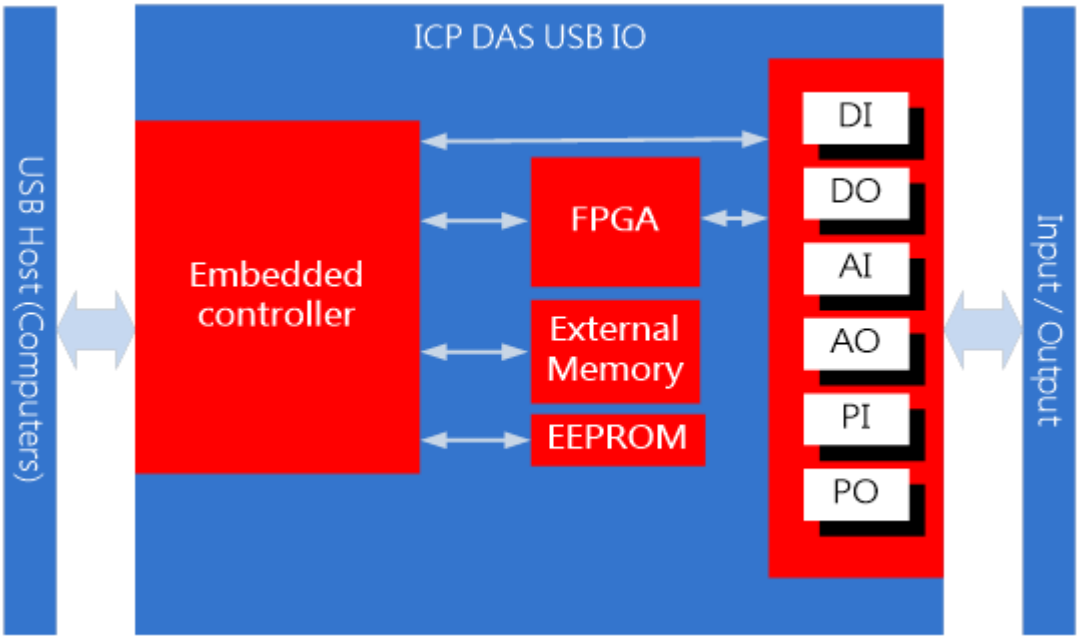
This chapter explains the operation and deployment process of the USB-2200 series. It covers the hardware and software architecture, SDK deployment, software structure, and class library operations, ensuring developers can quickly integrate the module into applications and use the APIs correctly.

2.1. Hardware Architecture

This section introduces the internal hardware design of the USB-2255, including its embedded architecture and I/O management components.

The USB-2200 series uses an embedded control architecture and connects to the host computer via USB. It consists of an embedded controller, FPGA, memory, and EEPROM, which manage I/O interfaces such as DI, DO, AI, AO, PI, and PO.

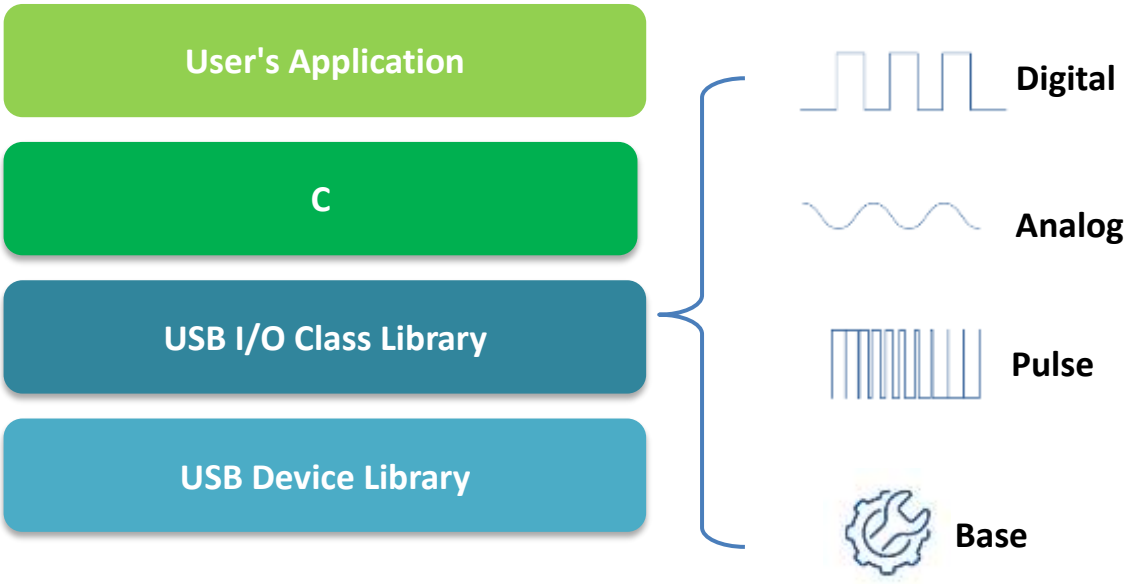
The following diagram illustrates the overall hardware block structure and signal flow of USB-2200 series.



2.2. Software Structure

This section explains the layered software architecture of the USB-2200 series, which separates device drivers, I/O class libraries, and user applications. By understanding this structure, developers can simplify integration and improve code maintainability.

The USB-2200 series adopts layered software architecture, comprising the "USB Device Driver Layer", "USB I/O Class Library Layer", and "User Application Layer". This modular design allows developers to easily integrate and operate I/O functions (DI/DO, AI/AO, PI/PO) using programming languages C. The architecture enhances code readability and maintainability while reducing integration barriers across platforms.



To help developers understand how to use the USB I/O class library, the following diagram presents the complete software flow—from device connection, I/O access, to device closure and resource release.

The software operation flow is structured into four key stages: (1) Open Device, (2) I/O Access, and (3) Close Device. Each stage corresponds to specific library methods and syntax examples. Detailed descriptions and implementation guidance are provided in the following subsections.

2.2.1. Open Device

Use USB2200_OpenDevice() to connect to the device.

USB2200_OpenDevice()

2.2.2. Access I/O

Once the connection is established, I/O access methods can be used to read or write various I/O channels such as DI, DO, AI and AO.

USB2200_DO_WriteValue

2.2.3. Close Device

After the application is finished, use USB2200_CloseDevice() to release system resources.

USB2200_CloseDevice

3. API Reference

This chapter describes the C application programming interface (API) for the USB-2200 series modules under Linux. The API provides a set of high-performance functions to perform device discovery, hardware configuration, and real-time I/O operations. All communication with the hardware is managed through a library handle, ensuring thread safety and efficient resource management.

API Classification
The USB-2200 C library functions are logically organized into the following categories:

1. Device Enumeration and Discovery
These functions allow the application to scan the USB bus and retrieve information about connected USB-2200 modules.

Scan and Identify: Locate all compatible devices currently attached to the system.

Information Retrieval: Obtain hardware details such as Model Name and Firmware Version.

2. Device Management (Session Control)
Before performing any I/O operations, a communication session must be established.

Open: Initialize the connection to a specific module and obtain a valid Device Handle.

Close: Terminate the session and release allocated system resources to ensure hardware availability for other processes.

3. I/O Control and Data Acquisition
Once a handle is obtained, these functions provide direct access to the module's hardware features:

Analog Input (AI): Read voltage/current signals with specified range and sampling settings.
Digital I/O (DI/DO): Monitor input states and control output channels.
Configuration: Modify internal hardware parameters such as gain, filter settings, and sampling rates.

3.1. Device Management and Identification

This section outlines the core functions required for fundamental module management and information retrieval. These functions enable developers to establish a communication session, synchronize data transfer, and query the specific hardware capabilities of the USB-2200 module.

3.1.1. USB2200_ListDevice

The USB2200_ListDevice function scans the USB bus and retrieves a list of all connected USB-2200 series modules. For each detected device, the function provides its **Product ID (PID)** and **Board ID (BID)**. This is a critical step for applications that need to identify specific hardware units when multiple modules are connected to the same host.

- **Product ID (PID):** Identifies the specific model of the USB-2200 series (e.g., distinguishing between a 4-channel and an 8-channel module).
- **Board ID (BID):** Identifies the hardware dip-switch setting on the module, allowing the software to differentiate between multiple units of the same model.

Syntax

```
int USB2200_ListDevice()
```

Parameter

Return Value

Error code

Example

Demo link

3.1.2. USB2200_OpenDevice

Open USB-2200 module with Product ID and Board ID.
You can check demo “list_device” to get Product ID and Board ID

Syntax

```
int USB2200_OpenDevice(
 WORD pid, BYTE bid
)
```

Parameter

pid
[IN] Product ID for the specific device to open.

bid
[IN] Board ID for the specific device to open.

Return Value

Error code.

Example

```
int handle;
WORD pid = USB2255_32;
BYTE bid = 0x0;

handle = USB2200_OpenDevice (pid, bid);
USB2200_CloseDevice (handle);
```

3.1.3. USB2200_CloseDevice

This function terminates the communication session with a specific USB-2200 module. This function ensures that all allocated system resources are properly released and the device becomes available for other processes.

Syntax

```
WORD USB2200_CloseDevice(
  int handle
)
```

Parameter

handle
[IN] A unique identifier returned by USB2200_OpenDevice() used to reference a specific module.

Return Value

Error code.

Example

```
int handle;
WORD pid = USB2255_32;
BYTE bid = 0x0;

handle = USB2200_OpenDevice (pid, bid);
USB2200_CloseDevice (handle);
```

3.1.4. USBIO_GetLibraryVersion

The USBIO_GetLibraryVersion function retrieves the current version information of the USB-2200 series shared library (SDK) installed on the Linux system. This is a utility function used to ensure compatibility between the application and the driver library.

Syntax

```
Char* USBIO_GetLibraryVersion ()
```

Parameter

Return Value

Error code.

Example

```
Demo link
```

3.1.5. USB2200_LoadDefault

Description The USB2200_LoadDefault function resets all internal configuration parameters of the module to their original factory settings. This includes restoring default I/O modes, gain settings, and communication parameters.

Important: This is an "Action" type command. After calling this function, a **hardware hot-plug (unplug and replug the USB cable)** is required for the new default settings to take effect and be reloaded by the module's firmware.

Syntax

```
WORD USB2200_LoadDefault (
  int handle
)
```

Parameter

handle

[IN] A unique identifier returned by USB2200_OpenDevice() used to reference a specific module.

Return Value

Error code.

Example

```
Demo link
```

3.1.6. USB2200_GetDeviceNickName

Retrieves the user-defined logical name (nickname) currently stored in the module's non-volatile memory. This string allows developers to identify and categorize specific hardware units using descriptive labels rather than static hardware IDs.

(Note: The nickname is limited to a maximum of 32 characters.)

Syntax

```
WORD USB2200_GetDeviceNickName(
  int handle, const char *desc
)
```

Parameter

`handle`

[IN] A unique identifier returned by USB2200_OpenDevice() used to reference a specific module.

desc

[OUT] A pointer to a character buffer that receives the retrieved nickname string.

(Note: The buffer must be at least 33 bytes to accommodate the maximum 32-character string plus the null terminator \0.)

Return Value

Error code.

Example

```
Demo link
```

3.1.7. USB2200_SetDeviceNickName

Retrieves the user-defined logical name (nickname) stored in the module's non-volatile memory. This allows for device identification by a custom string rather than hardware IDs.

*(Note: The nickname is limited to a maximum of **32 characters**.)*

Syntax

```
WORD USB2200_SetDeviceNickName(
  int handle, const char *desc
)
```

Parameter

`handle`

[IN] A unique identifier returned by USB2200_OpenDevice() used to reference a specific module.

`desc`

[IN] A pointer to a null-terminated string to be assigned as the device nickname.

Return Value

Error code.

Example

```
Demo link
```

3.1.8. USB2200_GetSoftWDTTimeout

Retrieves the current timeout value of the Software Watchdog Timer (WDT). The Watchdog is a safety mechanism that monitors the communication status between the host and the module; if the host fails to "kick" or reset the timer within this period, the module will enter a predefined safe state.

Note: The timeout value is typically defined in milliseconds (ms).

Syntax

```
WORD USB2200_GetSoftWDTTimeout (
  int handle, DWORD *timeout
)
```

Parameter

`handle`

[IN] A unique identifier returned by USB2200_OpenDevice() used to reference a specific module.

`timeout`

[OUT] A pointer to a DWORD variable that receives the current Software Watchdog timeout value.

Return Value

Error code.

Example

```
Demo link
```

3.1.9. USB2200_SetSoftWDTTimeout

Configures the expiration period for the Software Watchdog Timer (WDT). Once set, the module begins a countdown; the host application must periodically "reset" this timer to prevent it from reaching zero. If the timer expires (reaches zero) due to a software crash or communication loss, the module will trigger a safety response, such as deactivating all outputs and lighting the Watchdog Error LED.

Note: The timeout interval is specified in milliseconds (ms).

Valid Range: 100 ms to 1,800,000 ms (30 minutes).

Setting this value to 0 will disable the Watchdog function.

Syntax

```
WORD USB2200_SetSoftWDTTimeout (
  int handle, DWORD *timeout
)
```

Parameter

`handle`

[IN] A unique identifier returned by USB2200_OpenDevice() used to reference a specific module.

`timeout`

[OUT] The desired expiration period for the Software Watchdog Timer. Unit: Milliseconds (ms).

Return Value

Error code.

Example

```
Demo link
```

3.1.10. USB2200_SYNCDevice

Synchronizes and activates the operational state of the USB-2200 module. This function serves two critical purposes in the system's watchdog (WDT) lifecycle:

Activation: After calling USB2200_SetSoftWDTTimeout, you must call this function to officially start the WDT countdown. Without this synchronization, the watchdog remains idle.

Watchdog Refresh (Kicking the Dog): Once the WDT is running, this function acts as the "refresh" command. The host application must periodically call USB2200_SYNCDevice within the timeout period to reset the timer and prevent a safety trigger (Output deactivation).

Key Behavior:
Requirement: Must be called immediately after enabling WDT to begin counting.
Routine Task: Must be integrated into the main application loop to maintain continuous communication and avoid WDT timeouts.

Syntax

```
WORD USB2200_SYNCDevice (
  int handle
)
```

Parameter

`handle`
[IN] A unique identifier returned by USB2200_OpenDevice() used to reference a specific module.

Return Value

Error code.

Example

```
Demo link
```

3.1.11. USB2200_ClearWDTErrLED

Description
Clears the Watchdog Error state and deactivates the Watchdog Error LED on the module. This function is used to recover the device after a timeout event has occurred.

Behavior & Recovery:
Alarm Reset: Once a Watchdog timeout triggers (due to missing a SYNCDevice refresh), the module locks into a "Safe State" and lights the red Error LED.
Resuming Operations: Calling this function clears the error flag, turns off the LED, and allows the module to accept new output commands again.

Note: Ensure the root cause of the timeout (e.g., software hang or communication lag) is resolved before clearing the error to prevent immediate re-triggering.

Syntax

```
WORD USB2200_ClearWDTErrLED (
  int handle
)
```

Parameter

`handle`
[IN] A unique identifier returned by USB2200_OpenDevice() used to reference a specific module.

Return Value

Error code.

Example

```
Demo link
```

3.1.12. USB2200_GetFwVer

Retrieves the firmware version currently running on the USB-2200 module's internal microprocessor. This function is essential for verifying hardware features, ensuring compatibility with the SDK, and performing system diagnostics.

Syntax

```
WORD USB2200_GetFwVer(
  int handle, WORD *FwVer
)
```

Parameter

`handle`

[IN] A unique identifier returned by USB2200_OpenDevice() used to reference a specific module.

`FwVer`

[OUT] A pointer to a WORD variable that receives the firmware version of the module.

Data Format: The version is encoded as a Hexadecimal value.

Example: A returned value of 0x0120 should be interpreted as Version 1.20.

Return Value

Error code.

Example

```
Demo link
```

3.1.13. USB2200_GetFwDate

Retrieves the factory build date of the firmware currently running on the module .

Syntax

```
WORD USB2200_GetFwDate (
  int handle, char *FwDate
)
```

Parameter

`handle`

[IN] A unique identifier returned by USB2200_OpenDevice() used to reference a specific module.

`FwDate`

[OUT]A pointer to a character buffer (char *) that receives the firmware build date string.

Buffer Size: It is recommended to allocate at least 16 bytes to ensure enough space for the date and the null terminator.

Return Value

Error code.

Example

```
Demo link
```

3.1.14. USB2200_GetDOTotal

Get DO total number of channels of the device.

Syntax

```
WORD USB2200_GetDOTotal (
  int handle, BYTE *o_byDOTotal
)
```

Parameter

`handle`
[IN] A unique identifier returned by USB2200_OpenDevice() used to reference a specific module.
`o_byDOTotal`
[OUT] The DO total number of channels.

Return Value

Error code.

Example

```
Demo link
```

3.1.15. USB2200_GetDITotal

Get DI total number of channels of the device.

Syntax

```
WORD USB2200_GetDITotal (
  int handle, BYTE *o_byDITotal
)
```

Parameter

`handle`
[IN] A unique identifier returned by USB2200_OpenDevice() used to reference a specific module.
`o_byDITotal`
[OUT] The DI total number of channels.

Return Value

Error code.

Example

```
Demo link
```

3.1.16. USB2200_GetAOTotal

Get AO total number of channels of the device.

Syntax

```
WORD USB2200_GetAOTotal (
  int handle, BYTE *o_byAOTotal
)
```

Parameter

`handle`
[IN] A unique identifier returned by USB2200_OpenDevice() used to reference a specific module.
`o_byAOTotal`
[OUT] The AO total number of channels.

Return Value

Error code.

Example

```
Demo link
```

3.1.17. USB2200_GetAITotal

Get AI total number of channels of the device.

Syntax

```
WORD USB2200_GetAITotal (
  int handle, BYTE *o_byAITotal
)
```

Parameter

`handle`
[IN] A unique identifier returned by USB2200_OpenDevice() used to reference a specific module.
`o_byAITotal`
[OUT] The AI total number of channels.

Return Value

Error code.

Example

```
Demo link
```

3.1.18. USB2200_GetPOTotal

Get PO total number of channels of the device.

Syntax

```
WORD USB2200_GetPOTotal (
  int handle, BYTE *o_byPOTotal
)
```

Parameter

`handle`
[IN] A unique identifier returned by USB2200_OpenDevice() used to reference a specific module.
`o_byPOTotal`
[OUT] The PO total number of channels.

Return Value

Error code.

Example

```
Demo link
```

3.1.19. USB2200_GetPITotal

Get PI total number of channels of the device.

Syntax

```
WORD USB2200_GetPITotal (
  int handle, BYTE *o_byPITotal
)
```

Parameter

`handle`
[IN] A unique identifier returned by USB2200_OpenDevice() used to reference a specific module.
`o_byPITotal`
[OUT] The PI total number of channels.

Return Value

Error code.

Example

```
Demo link
```

3.2. Digital Output

This section describes methods for Digital Output functionality. It includes digital output read/write, power-on settings, safety value configuration, and inversion control, enabling digital control and protection logic.

3.2.1. USB2200_DO_WriteValue

Writes the output states to **all Digital Output (DO) channels** simultaneously using a single bitmask value.

Enable Bit	Value
Off	0
On	1

Syntax

```
WORD USB2200_DO_WriteValue (
  int handle, QWORD val
)
```

Parameter

`handle`

[IN] A unique identifier returned by USB2200_OpenDevice() used to reference a specific module.

`val`

[IN]A bitmask representing the desired state of all DO channels.

Logic: Each bit corresponds to a specific channel (e.g., Bit 0 = Channel 0).

State: 1 = ON (Active), 0 = OFF (Inactive).

Return Value

Error code.

Example

```
Demo link
```

3.2.2. USB2200_DO_WriteChannel

Updates the output state of a **single Digital Output (DO) channel** without affecting the current states of other channels.

Enable Bit	Value
Off	0
On	1

Syntax

```
WORD DO_WriteValue(
  int handle, BYTE channel, BYTE val
)
```

Parameter

`handle`

[IN] A unique identifier returned by USB2200_OpenDevice() used to reference a specific module.

`channel`

[IN]The specific DO channel index to be modified (e.g., 0, 1, 2...).

`val`

The desired output state for the selected channel. **1:** ON (Active) **0:** OFF (Inactive)

Return Value

Error code.

Example

```
Demo link
```

3.2.3. USB2200_DO_ReadValue

Retrieves the current output states of **all Digital Output (DO) channels** from the hardware.

Syntax

```
WORD DO_ReadValue(
  int handle, QWORD *val
)
```

Parameter

`handle`

[IN] A unique identifier returned by USB2200_OpenDevice() used to reference a specific module.

`val`

[OUT]A pointer to a QWORD variable that receives the bitmask of all DO states

Logic: Each bit represents a channel (e.g., Bit 0 = Channel 0).
State: 1 = ON, 0 = OFF.

Return Value

Error code.

Example

```
Demo link
```

3.2.4. USB2200_DO_SetPowerOnEnable

Configures the **default output states** for all Digital Output (DO) channels immediately upon device power-up. This ensures the hardware starts in a known, safe configuration before the host application sends its first command.

Syntax

```
WORD USB2200_DO_SetPowerOnEnable (
  int handle, QWORD val
)
```

Parameter

`handle`
[IN] A unique identifier returned by USB2200_OpenDevice() used to reference a specific module.

`val`
[IN]A bitmask representing the power-on state for all DO channels.

1: ON (Active) upon power-up.
0: OFF (Inactive) upon power-up.

Return Value

Error code.

Example

```
Demo link
```

3.2.5. USB2200_DO_SetPowerOnEnableToChannel

Sets the power-on default state for a **single specific Digital Output (DO) channel**. This configuration is stored in the hardware's non-volatile memory and takes effect immediately upon the next power cycle or device reset.

Syntax

```
WORD USB2200_DO_SetPowerOnEnableToChannel (
  int handle, BYTE ch, BYTE bit
)
```

Parameter

`handle`

[IN] A unique identifier returned by USB2200_OpenDevice() used to reference a specific module.

`ch`

[IN]The target DO channel index to be configured (e.g., 0, 1, 2 ... 15).

`bit`

[IN] The desired power-on state for the selected channel.

1: Enable (Output ON at power-up).
0: Disable (Output OFF at power-up).

Return Value

Error code.

Example

```
Demo link
```

3.2.6. USB2200_DO_GetPowerOnEnable

Retrieves the current power-on default configuration for **all Digital Output (DO) channels** from the module's non-volatile memory. This is used to verify the stored hardware-level safety states.

Power-On Enable	Value
Disable & Off	0
Enable & On	1

Syntax

```
WORD USB2200_DO_GetPowerOnEnable (
  int handle, QWORD *val
)
```

Parameter

`handle`

[IN] A unique identifier returned by USB2200_OpenDevice() used to reference a specific module.

`val`

[IN]A pointer to a QWORD variable that receives the bitmask of the power-on states.

Logic: Each bit corresponds to a specific channel (e.g., Bit 0 = Channel 0).

State: 1 = Enabled (ON at power-up), 0 = Disabled (OFF at power-up).

Return Value

Error code.

Example

```
Demo link
```

3.2.7. USB2200_DO_SetSafetyEnable

Configures the **Safety State (Fail-Safe)** for Digital Output channels. This defines how the hardware responds when a communication error occurs or the Watchdog Timer (WDT) expires. These settings are stored in the module's non-volatile memory.

Syntax

```
WORD USB2200_DO_SetSafetyEnable (
  int handle, QWORD enable_mask, QWORD val
)
```

Parameter

`handle`

[IN] A unique identifier returned by USB2200_OpenDevice() used to reference a specific module.

`enable_mask`

[IN] A bitmask to enable/disable the safety function for each channel.

1: Enable Safety Mode for this channel.

0: Disable Safety Mode (Channel remains in its last known state).

`val`

[IN] A bitmask representing the Safety Value to be applied when the safety function is triggered.

1: Force Output ON.

0: Force Output OFF.

Return Value

Error code.

Example

> Demo link

3.2.8. USB2200_DO_SetSafetyEnableChannel

Configures the **Safety (Fail-Safe) settings** for a single specific Digital Output (DO) channel. This determines if the individual channel should enter a predefined state when a communication error or Watchdog Timer (WDT) timeout occurs.

Syntax

```
WORD USB2200_DO_SetSafetyEnableChannel (
  int handle, BYTE ch, BYTE enable, BYTE val
)
```

Parameter

`handle`

[IN] A unique identifier returned by USB2200_OpenDevice() used to reference a specific module.

`ch`

[IN] The target DO channel index (e.g., 0 to 15).

`enable`

[IN] Enables or disables the safety function for this specific channel.

1: Enable (The channel will switch to the val state upon failure).

0: Disable (The channel will maintain its current state upon failure).

`bit`

[IN] The designated Safety Value for the channel.

1: Force Output High/ON.

0: Force Output Low/OFF.

Return Value

Error code.

Example

> Demo link

3.2.9. USB2200_DO_GetSafetyEnable

Retrieves the current Safety (Fail-Safe) configuration for all Digital Output (DO) channels from the module's non-volatile memory. This allows the application to verify which channels have safety protection enabled and their corresponding failsafe output values.

Syntax

```
WORD USB2200_DO_GetSafetyEnable (
  int handle, QWORD *enable_mask, QWORD *val
)
```

Parameter

`handle`

[IN] A unique identifier returned by USB2200_OpenDevice() used to reference a specific module.

`enable_mask`

[OUT] A pointer to a QWORD variable that receives the safety enable bitmask.

1: Safety protection is Enabled for this channel.

0: Safety protection is Disabled (Channel maintains last state).

`val`

[OUT] A pointer to a QWORD variable that receives the predefined Safety Value bitmask.

1: Channel will force to High/ON upon failure.

0: Channel will force to Low/OFF upon failure.

Return Value

Error code.

Example

> Demo link

3.2.10. USB2200_DO_SetInverse

Configures the global output logic for all Digital Output (DO) channels. This setting determines whether a software command of 1 results in a physical High or Low voltage level.

Inverse	Value
No	0
Yes	1

Syntax

```
WORD USB2200_DO_SetInverse(
  int handle, BYTE val
)
```

Parameter

`handle`

[IN] A unique identifier returned by USB2200_OpenDevice() used to reference a specific module.

`val`

[IN] The desired logic mode for all DO channels.

0 (Normal): Active High. (Software 1 = Physical High; Software 0 = Physical Low).

1 (Inverted): Active Low. (Software 1 = Physical Low; Software 0 = Physical High).

Return Value

Error code.

Example

```
Demo link
```

3.2.11. USB2200_DO_GetInverse

Retrieves the current global output logic setting for all Digital Output (DO) channels from the module's non-volatile memory. This is used to verify whether the system is operating in Normal or Inverted logic mode.

Inverse	Value
No	0
Yes	1

Syntax

```
WORD USB2200_DO_GetInverse (
  int handle, BYTE *val
)
```

Parameter

`handle`

[IN] A unique identifier returned by USB2200_OpenDevice() used to reference a specific module.

`val`

[OUT] A pointer to a BYTE variable that receives the current logic mode.

0 (Normal): Active High logic is currently in use.

1 (Inverted): Active Low logic is currently in use.

Return Value

Error code.

Example

```
Demo link
```

3.3. Digital Input

This section introduces methods for Digital Input functionality. It includes digital value reading, filter configuration, input inversion, edge triggering, and counter operations for typical DI data acquisition.

3.3.1. USB2200_DI_ReadValue

Reads the current input states of **all Digital Input (DI) channels** simultaneously. This function retrieves the real-time logical status (High/Low) from the hardware sensing circuitry.

Syntax

```
WORD USB2200_DI_ReadValue(
  int handle, QWORD *val
)
```

Parameter

`handle`

[IN] A unique identifier returned by USB2200_OpenDevice() used to reference a specific module.

val

[OUT] A pointer to a QWORD variable that receives the bitmask of all DI states.

Logic: Each bit corresponds to a specific input channel (e.g., Bit 0 = DI Channel 0).

State: 1 = High/Active, 0 = Low/Inactive.

Return Value

Error code.

Example

```
Demo link
```

3.3.2. USB2200_DI_ReadChannel

Reads the real-time logical state of a **single specified Digital Input (DI) channel**. This function simplifies the application logic by isolating the target channel's status from the full input bitmask.

Syntax

```
WORD USB2200_DI_ReadChannel (
  int handle, BYTE ch, BYTE *val
)
```

Parameter

`handle`

[IN] A unique identifier returned by USB2200_OpenDevice() used to reference a specific module.

`ch`

[IN] The target DI channel index to be read (e.g., 0 to 15).

`val`

[OUT] (Output): A pointer to a BYTE variable that receives the channel's current state.

1: The input signal is High/Active.

0: The input signal is Low/Inactive.

Return Value

Error code.

Example

```
Demo link
```

3.3.3. USB2200_DI_SetFilter

Sets the **Hardware Digital Filter** width for all Digital Input (DI) channels. This function defines the minimum duration an input signal must remain stable to be recognized as a valid logical state. It is essential for eliminating high-frequency noise and mechanical contact chatter (debounce).The unit of the filter width is 0.1 million-second..

Syntax

```
WORD USB2200_DI_SetFilter (
  int handle, WORD val
)
```

Parameter

`handle`

[IN] A unique identifier returned by USB2200_OpenDevice() used to reference a specific module.

`val`

[IN] Filter interval value.

Unit: 0.1 ms (100 microseconds) per step.

Range: 0 to 65535.

Setting 0: Disables the filter (Maximum sensitivity).

Return Value

Error code.

Example

```
Demo link
```

3.3.4. USB2200_DI_GetFilter

Retrieves the current **Hardware Digital Filter** width setting for all Digital Input (DI) channels from the module's non-volatile memory. This allows the application to verify the existing debounce interval used by the hardware sensing circuitry. The unit of the filter is 0.1 million-second.

Syntax

```
WORD USB2200_DI_GetFilter (
  int handle, WORD *val
)
```

Parameter

`handle`

[IN] A unique identifier returned by USB2200_OpenDevice() used to reference a specific module.

`val`

[OUT] A pointer to a WORD variable that receives the filter interval value.

Unit: 0.1 ms (100 microseconds).

State: 0 means the filter is disabled.

Calculation: Real-time filter width (ms) = *val × 0.1.

Return Value

Error code.

Example

```
Demo link
```

3.3.5. USB2200_DI_SetInverse

Sets the global input logic polarity for all Digital Input (DI) channels.

Inverse	Value
No	0
Yes	1

Syntax

```
WORD USB2200_DI_SetInverse (
  int handle, BYTE val
)
```

Parameter

`handle`

[IN] A unique identifier returned by USB2200_OpenDevice() used to reference a specific module.

`val`

[IN] 0: Normal (Active High).

1: Inverted (Active Low).

Return Value

Error code.

Example

```
Demo link
```

3.3.6. USB2200_DI_GetInverse

Retrieves the current global input logic polarity for all Digital Input (DI) channels from the module's non-volatile memory. This function is used to verify whether the system is interpreting physical signals in **Normal** or **Inverted** mode.

Inverse	Value
No	0
Yes	1

Syntax

```
WORD USB2200_DI_GetInverse (
  int handle, BYTE *val
)
```

Parameter

`handle`

[IN] A unique identifier returned by USB2200_OpenDevice() used to reference a specific module.

`val`

[OUT] A pointer to a BYTE variable that receives the current logic mode.

0 (Normal): Active High logic is currently in use. (High voltage = 1).

1 (Inverted): Active Low logic is currently in use. (Low voltage = 1).

Return Value

Error code.

Example

```
Demo link
```

3.3.7. USB2200_DI_SetCntEdgeTrigger

Configures the **Edge Trigger** condition for all hardware-based Digital Input (DI) counters. This function determines whether the counter increments on the leading edge (Rising) or trailing edge (Falling) of the input signal. This configuration is stored in the module's non-volatile memory.

Edge Trigger	Value
Falling	0
Rising	1

Syntax

```
WORD USB2200_DI_SetCntEdgeTrigger(
  int handle, BYTE val
)
```

Parameter

`handle`

[IN] A unique identifier returned by USB2200_OpenDevice() used to reference a specific module.

`Val`

[IN] The desired edge trigger mode.

0 (Rising Edge): Increments when the signal transitions from Low to High.

1 (Falling Edge): Increments when the signal transitions from High to Low.

Return Value

Error code.

Example

```
Demo link
```

3.3.8. USB2200_DI_GetCntEdgeTrigger

Retrieves the current **Edge Trigger** configuration for all hardware Digital Input (DI) counters from the module's non-volatile memory. This allows the application to verify if the counters are incrementing on a **Rising Edge** or a **Falling Edge** transition.

Edge Trigger	Value
Falling	0
Rising	1

Syntax

```
WORD USB2200_DI_GetCntEdgeTrigger (
  int handle, BYTE *val
)
```

Parameter

`handle`

[IN] A unique identifier returned by USB2200_OpenDevice() used to reference a specific module.

`val`

[OUT] A pointer to a BYTE variable that receives the current trigger mode.

0 (Rising Edge): Counters increment on Low-to-High transitions.

1 (Falling Edge): Counters increment on High-to-Low transitions.

Return Value

Error code.

Example

Demo link

3.3.9. USB2200_DI_ReadCounterValue

Retrieves the current accumulated count values for a specified range of Digital Input (DI) Hardware Counters. This function reads the 32-bit internal registers that track pulse transitions based on the configured edge trigger.

Syntax

```
WORD USB2200_DI_ReadCounterValue (
  int handle, DWORD *counter_array, BYTE di_num
)
```

Parameter

`handle`

[IN] A unique identifier returned by USB2200_OpenDevice() used to reference a specific module.

`counter_array`

[OUT] A pointer to a DWORD array (unsigned 32-bit) that will receive the counter values.

The array size must be at least di_num elements.

`di_num`

[IN] The number of channels to read, starting from DI channel 0.

Example: If di_num is 16, the API will fill the array with values from DI0 to DI15.

Return Value

Error code.

Example

```
Demo link
```

3.3.10. USB2200_DI_ReadChannelCounter

Retrieves the current 32-bit hardware counter value for a **single specified Digital Input (DI) channel**. This function is ideal for applications that only need to monitor one specific pulse source, such as a single flow meter or an encoder index, without the overhead of reading all channels.

Syntax

```
WORD USB2200_DI_ReadChannelCounter (
  int handle, BYTE ch, DWORD *val
)
```

Parameter

`handle`

[IN] A unique identifier returned by USB2200_OpenDevice() used to reference a specific module.

`ch`

[IN] The target DI channel index to be read (e.g., 0 to 15).

`val`

[OUT] A pointer to a DWORD variable that receives the 32-bit accumulated count value for the specified channel.

Return Value

Error code.

Example

```
Demo link
```

3.3.11. USB2200_DI_ClearCounter

Resets the specified 32-bit hardware counters to zero. This function uses a **64-bit mask (QWORD)** to identify which DI channels should be cleared. Each bit in the mask corresponds to a specific DI channel; setting a bit to 1 will trigger a reset for that channel's counter.

Syntax

```
WORD USB2200_DI_ClearCounter (
  int handle, QWORD clear_mask
)
```

Parameter

`handle`

[IN] A unique identifier returned by USB2200_OpenDevice() used to reference a specific module.

`clear_mask`

[IN] A 64-bit unsigned integer (QWORD) representing the channels to be cleared.

Bit 0 (0x1): Clears DI Channel 0.

Bit 15 (0x8000): Clears DI Channel 15.

Value 0xFFFFFFFF: Clears all available DI counters.

Return Value

Error code.

Example

```
Demo link
```

3.3.12. USB2200_DI_ClearCounterChannel

Resets the 32-bit hardware counter of a **single specified Digital Input (DI) channel** to zero. This function is a simplified alternative to the mask-based clear, allowing developers to target one specific input without calculating bitmasks.

Syntax

```
WORD USB2200_DI_ClearCounterChannel (
  int handle, BYTE channel
)
```

Parameter

`handle`
[IN] A unique identifier returned by USB2200_OpenDevice() used to reference a specific module.
`channel`
[IN] The index of the DI channel to be cleared (e.g., 0 to 15).
Example: Passing 15 will only reset the counter for DI Channel 15, leaving all other 15 channels' values intact.

Return Value

Error code.

Example

```
Demo link
```

4. Demo program for linux

To help developers quickly integrate USB-2200 series modules into their systems, this chapter provides a collection of C-based demo programs. These examples are designed to demonstrate the core functionalities of the API, ranging from basic device identification to advanced data acquisition. Each demo serves as a practical reference for implementing stable communication and efficient I/O control in a Linux environment.

4.1. list_device

The list_device demo is a fundamental utility designed to verify the connection status between the Linux host and the USB-2200 series hardware. By invoking the USB2200_ListDevice API, this program scans the system's USB bus and displays a list of all detected modules.

For each discovered device, the demo prints the following identifiers:

- **Product ID (PID):** Used to confirm the specific model of the module.
- **Board ID (BID):** Used to distinguish between multiple units of the same model based on their hardware dip-switch settings.

This demo serves as the first step in troubleshooting and system configuration, ensuring that the driver and hardware are communicating correctly before proceeding to data acquisition tasks.

```
root@icpdas:~/usb-2200/examples# ./list_device
Found 2 USB-2200 device(s):
 [0] PID=USB2255_64  BID=0x2
 [1] PID=USB2255_32  BID=0x0
```

If you want to open USB-2255-32 , then the parameter PID is USB2255_32, BID is 0

4.2. module_setting

The module_setting demo is a comprehensive configuration utility designed for system administrators and developers to manage the internal parameters of a USB-2200 module. It provides an interactive command-line interface (CLI) to perform hardware diagnostics, retrieve device metadata, and adjust critical safety settings.

Key Functionalities The demo program allows users to perform the following operations:

- **Device Identification:** Retrieve firmware versions, update dates, and library details.
- **Hardware Profiling:** Query the total number of available channels for all I/O types (AI, AO, DI, DO, PI, and PO).
- **System Configuration:** Customize the device description name and manage **Watchdog (WDT)** timeout values for safety monitoring.
- **Administrative Actions:** * **Factory Reset:** Restore the module to its default factory settings (requires a hardware hot-plug to take effect).
    - **SYNC Device:** Synchronize communication parameters between the host and the module.
    - **Maintenance:** Reset hardware error states, such as clearing the Watchdog Error LED.

```
--- USB-2200 Module System Configuration ---
1. [Get    ] Library Version
2. [Action ] Factory Reset (Set Module Default)
3. [Get/Set] Device Description Name
4. [Get/Set] Watchdog Timeout Value
5. [Action ] SYNC Device
6. [Action ] Clear Watchdog Error LED
7. [Get    ] Firmware Version.
8. [Get    ] Firmware Update Date.
9. [Get    ] DO Total Channel.
10.[Get    ] DI Total Channel.
11.[Get    ] AO Total Channel.
12.[Get    ] AI Total Channel.
13.[Get    ] PO Total Channel.
14.[Get    ] PI Total Channel.
-----------------------------------------
Others show this list, Q to return to main menu.
Select an option:
Press ESC to exit.
```

4.2.1. Library Version

```
1
--- Get Library Version ---
USB-2200 Library Version: 1.0.0
```

4.2.2. Factory Reset

This function resets all **Digital Input (DI) configurations** to their factory default states. It is a vital safety and initialization tool used to ensure the module starts from a known, "clean" configuration before any application-specific logic is applied.

```
2
--- Load to Default Setting ---

Press ESC to exit.
```

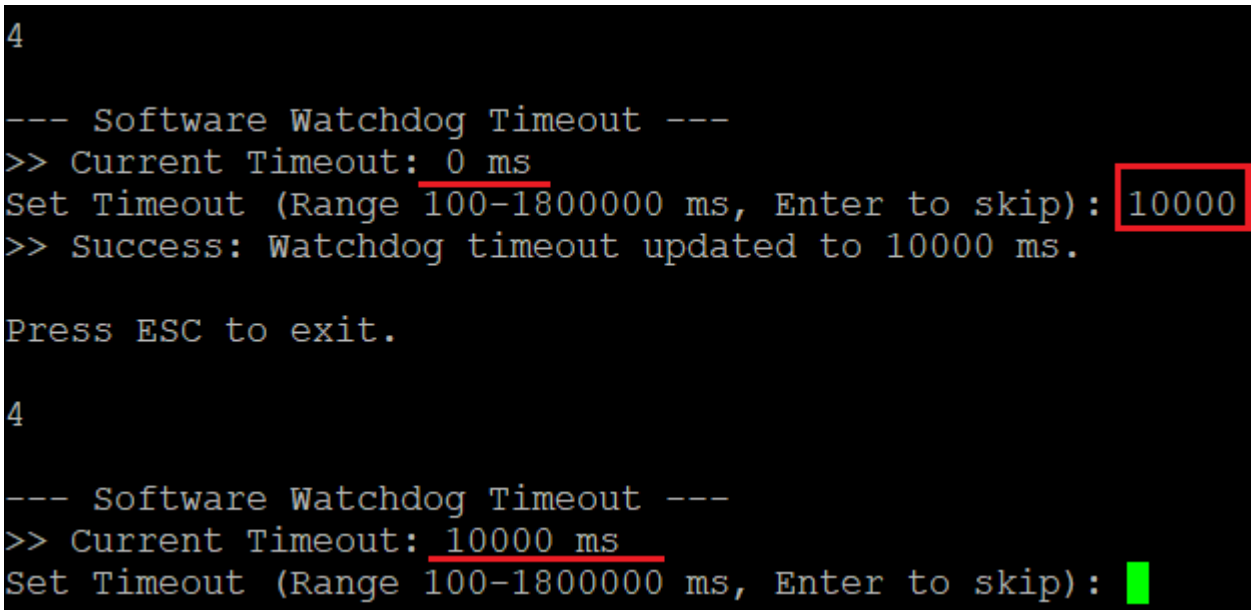
4.2.3. Device Description Name

The USB-2200 series features a programmable **Device Description Name** field. This allows users to assign a unique, user-defined string (up to 32 characters) to each module, which is stored in the device's internal memory.

```
3

--- Device Description Name Configuration ---
>> Current Name: "USB-2255-32 16DI/16DO"
Enter new name (Max 32 chars, Enter to skip): ICPDAS
>> Success: Device name updated.

Press ESC to exit.

3

--- Device Description Name Configuration ---
>> Current Name: "ICPDAS"
Enter new name (Max 32 chars, Enter to skip):
```

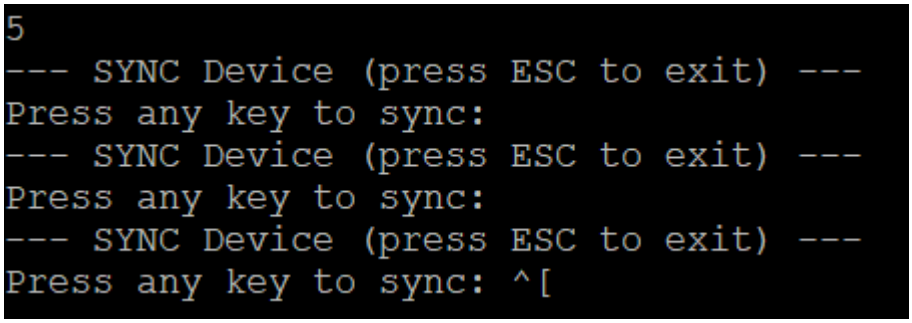
4.2.4. Watchdog Timeout Value

The **Software Watchdog** is a critical safety feature designed to monitor the communication health between the host PC and the USB-2200 module. If the host fails to "kick" (reset) the watchdog timer within the specified period—due to a software crash or cable disconnection—the module will automatically trigger a **Safety Value** for all outputs to prevent equipment damage.



4.2.5. SYNC Device

Description The **SYNC Device** function is the operational phase of the Software Watchdog system. Once a timeout period is configured, the host application must periodically send a **"Refresh" (or "Kick")** command to the module. This signal informs the hardware that the software is functioning correctly and the communication link is active.



4.2.6. Clear Watchdog Error LED

When a Watchdog Timeout occurs, the USB-2200 module enters Safety Mode and activates the physical Error LED. Even after communication is restored, the LED remains lit as a "latch" to notify operators of the previous failure. This Demo demonstrates how to clear this error flag and reset the LED indicator.

```
6
--- Clear WDT Err LED ---

Press ESC to exit.
```

4.2.7. Firmware Version

```
7

--- Firmware Information ---
>> Firmware Version: 0.60

Press ESC to exit.
```

4.2.8. Firmware Update Date

```
8

--- Firmware Release Date ---
>> Firmware Date: 20240117-0

Press ESC to exit.
```

4.2.9. DO Total Channel

```
9
Get DO total channel:16

Press ESC to exit.
```

4.2.10. DI Total Channel

```
10
Get DI total channel:16

Press ESC to exit.
```

4.2.11. AO Total Channel

```
11
Get AO total channel:0

Press ESC to exit.
```

4.2.12. AI Total Channel

```
12
Get AI total channel:0

Press ESC to exit.
```

4.2.13. PO Total Channel

```
13
Get PO total channel:0

Press ESC to exit.
```

4.2.14. PI Total Channel

```
14
Get PI total channel:0

Press ESC to exit.
```

4.3. Setdo

The setdo utility is a command-line tool designed to update the output states of all Digital Output (DO) channels simultaneously. It supports both decimal and hexadecimal input formats for flexible control.

Command Usage

Command: ./setdo <value>

Parameter <value>: * Decimal (e.g., 123): Useful for quick manual testing.

Hexadecimal (e.g., 0xFF): Preferred for mapping specific bits to DI channels (e.g., 0x01 for Channel 0, 0x80 for Channel 7).

Operational Analysis

Execution: root@icpdas:~/usb-2200/examples# ./setdo 0xff
Input Value: 0xff (Hexadecimal) = 1111 1111 (Binary).
Result: Read DO value 0xff

The system successfully confirmed that the output registers for the first 8 channels (DO 0 to DO 7) have been set to High (1).

Physical LEDs on the module for these channels will illuminate, and the corresponding relays or transistors will be triggered.

```
root@icpdas:~/usb-2200/examples# ./setdo
Usage: ./setdo <value>
  value: decimal (123) or hex (0xFF)
root@icpdas:~/usb-2200/examples# ./setdo 0xff
Read DO value 0xff
```

4.4. setdo_channel

The setdo_channel utility provides granular control over the Digital Output (DO) module. It allows users to toggle the state of a specific channel without affecting the current output status of any other channels.

Command Usage

Command: ./setdo_channel <channel> <value>

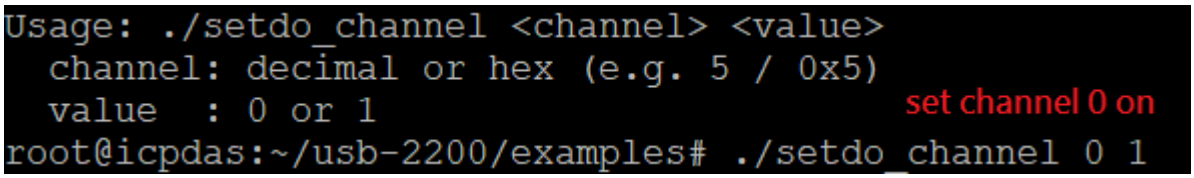
Parameter <channel>: The target DO channel index (e.g., 0 to 15). Supports both decimal and hexadecimal formats.

Parameter <value>: The desired output state.
1: Set channel to High/On.
0: Set channel to Low/Off.

Operational Analysis

Execution: root@icpdas:~/usb-2200/examples# ./setdo_channel 15 1
Target: Channel 15 (The last DO channel).
Action: Set to state 1 (High).

Result: Only the DO 15 hardware output is activated. This demonstrates the "Individual Control" capability, ensuring that channels 0 through 14 remain in their previous states.



4.5. getdo

The getdo utility allows the user to retrieve the current output states of all Digital Output (DO) channels in a single operation. This "read-back" capability is essential for verifying that the software commands have been correctly executed by the hardware and for monitoring the system's current state.

Operational Analysis

Execution: root@icpdas:~/usb-2200/examples# ./getdo

Output Result: Read DO value 0x80ff

Hexadecimal Breakdown:
0x80ff (Hex) = 1000 0000 1111 1111 (Binary).

Channel Status Mapping:
Channels 0–7: All are High (1).
Channels 8–14: All are Low (0).
Channel 15: Is High (1).

Key Features

Read-back Mechanism: It retrieves the state from the output latch register within the USB-2200 module, ensuring the data reflects the actual hardware configuration.

Synchronization Check: Frequently used in control loops to ensure the intended output matches the actual output before proceeding to the next process step.

```
root@icpdas:~/usb-2200/examples# ./getdo
Read DO value 0x80ff
```

4.6. do_setting

The do_setting demonstration illustrates how to configure the hardware-level parameters of the Digital Output (DO) channels. These settings are stored in the module's non-volatile memory, ensuring consistent behavior during power-up, communication loss, or specific logic requirements.

1. DO Power-On Status (Bulk/Channel)
Function: Defines the initial state of all DO channels immediately after the module receives power, before any software commands are issued.

Purpose: Ensures connected hardware remains in a safe, predictable state during the system boot sequence.

2. DO Safety Status & Value (Bulk/Channel)
Function: Configures the "Fail-Safe" state that the module automatically enters if a Watchdog Timer (WDT) timeout occurs or if the USB communication is interrupted.

Purpose: Provides hardware-level protection to prevent hazardous conditions in the event of a host computer crash or cable disconnection.

3. DO Inverse Mode (Normal/Inverted)
Function: Switches the output logic between Normal (Active High) and Inverted (Active Low).

Purpose: Allows the software to maintain a consistent logic regardless of whether the physical circuit requires a high or low voltage to trigger.

```
--- USB-2200 Digital Output Configuration ---
1. [Get/Set] DO PowerOn Status (Bulk/Channel)
2. [Get/Set] DO Safety Status & Value (Bulk/Channel)
3. [Get/Set] DO Inverse Mode (Normal/Inverted)
Others number show this list, ESC to quit.
Select an option:
Press ESC to exit.
```

4.6.1. DO PowerOn Status

This section demonstrates the step-by-step process of configuring the power-on default states for Digital Output (DO) channels. Users can choose between updating all channels at once (Bulk) or modifying a specific channel individually.

Scenario 1: Bulk Configuration (Set All by Mask)
Current State: The initial Power-On Mask is 0x0000000000000000 (All OFF).
Operation: Select Action (1) to update the entire bitmask.
Input: Enter a hex value (e.g., 0xFF).
Result: Channels 0 through 7 are now set to "Enable" upon power-up. The mask updates to 0x00000000000000FF.

Scenario 2: Single Channel Configuration
Current State: Power-On Mask is 0x00000000000000FF.
Operation: Select Action (2) to modify a specific channel without affecting others.
Input: Target Channel 15 and set its value to 1 (Enable).
Result: Channel 15 is activated in the hardware registry. The final Power-On Mask reflects the change as 0x00000000000080FF.

```
1

--- DO PowerOn Configuration ---
>> Current PowerOn Mask: 0x0000000000000000
Action: (0) Skip, (1) Set All by Mask, (2) Set Single Channel: 1
Enter new PowerOn Mask (e.g. 0xFF): 0xff
>> Bulk PowerOn updated.

Press ESC to exit.

1

--- DO PowerOn Configuration ---
>> Current PowerOn Mask: 0x00000000000000FF
Action: (0) Skip, (1) Set All by Mask, (2) Set Single Channel: 2
Enter Channel (0-15): 15
Set Value for Ch 15 (0:Disable, 1:Enable): 1
>> Channel 15 PowerOn updated.

Press ESC to exit.

1

--- DO PowerOn Configuration ---
>> Current PowerOn Mask: 0x00000000000080FF
Action: (0) Skip, (1) Set All by Mask, (2) Set Single Channel:
```

4.6.2. DO Safety Status & Value

This section demonstrates how to configure the Safety Enable Mask and the Safety Value Mask. These parameters determine which channels will react during a system failure (such as a Watchdog timeout) and what their predetermined "Safe State" will be.

Scenario 1: Bulk Safety Configuration
Initial State: Both Safety Enable and Value masks are 0x0000000000000000 (Safety mode is disabled for all channels).

Operation: Select Action (1) to update both masks simultaneously using hex values.
Input: * Enable Mask: 0xFF (Enable safety protection for Channels 0-7).
Value Mask: 0xFF (Set the failsafe state of Channels 0-7 to High/ON).

Result: If a communication failure occurs, Channels 0-7 will automatically switch to ON.

Scenario 2: Single Channel Safety Configuration
Current State: Enable and Value masks are 0x00000000000000FF.

Operation: Select Action (2) to modify a specific channel's safety behavior.
Input: * Channel: 15.
Enable: 1 (Enable safety for Ch 15).
Value: 1 (Set failsafe state for Ch 15 to High).

Result: The masks are updated to 0x00000000000080FF. Channel 15 is now protected and will trigger a High output upon system failure.

```
2

--- DO Safety Configuration (Failsafe State) ---
>> Current Safety Enable Mask: 0x0000000000000000
>> Current Safety Value  Mask: 0x0000000000000000

Action: (0) Skip, (1) Set Bulk Mask, (2) Set Single Channel: 1
Enter Enable Mask (Hex, e.g. 0xFF): 0xff
Enter Value Mask (Hex, e.g. 0x00): 0xff
>> Bulk Safety settings updated successfully.

Press ESC to exit.

2

--- DO Safety Configuration (Failsafe State) ---
>> Current Safety Enable Mask: 0x00000000000000FF
>> Current Safety Value  Mask: 0x00000000000000FF

Action: (0) Skip, (1) Set Bulk Mask, (2) Set Single Channel: 2
Enter Channel (0-15): 15
Safety Enable for Ch 15 (0:Disable, 1:Enable): 1
Safety Value for Ch 15 (0:Low, 1:High): 1
>> Channel 15 Safety updated successfully.

Press ESC to exit.

2

--- DO Safety Configuration (Failsafe State) ---
>> Current Safety Enable Mask: 0x00000000000080FF
>> Current Safety Value  Mask: 0x00000000000080FF

Action: (0) Skip, (1) Set Bulk Mask, (2) Set Single Channel:
```

4.6.3. DO Inverse Mode

This section demonstrates how to toggle the output logic of the Digital Output (DO) channels between **Normal** and **Inverted** modes. This setting globally redefines the relationship between the software bit value (0 or 1) and the physical voltage output.

Scenario: Switching from Normal to Inverted
Initial State: The current DO logic is set to [0] NORMAL.

Behavior: Software 1 = Physical ON (High); Software 0 = Physical OFF (Low).

Operation: Select Action (3) and enter 1 to switch to Inverse mode.

Input: 1 (Inverse).

Result: The logic is successfully updated to [1] INVERTED.

New Behavior: Software 1 = Physical OFF (Low); Software 0 = Physical ON (High).

```
3

--- DO Logic Inverse Mode (Normal/Inverted) ---
>> Current DO Logic: [0] NORMAL
Change setting? (0: Normal, 1: Inverse, Enter to Skip): 1
>> Success: DO Inverse updated to [Inverted]

Press ESC to exit.

3

--- DO Logic Inverse Mode (Normal/Inverted) ---
>> Current DO Logic: [1] INVERTED
Change setting? (0: Normal, 1: Inverse, Enter to Skip):
```

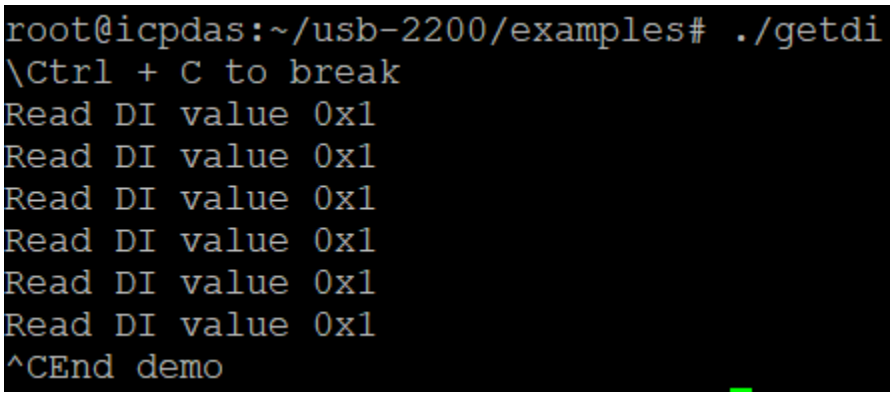
4.7. getdi

The getdi demonstration provides a real-time monitoring interface for all Digital Input (DI) channels. It continuously polls the hardware and displays the current input state as a hexadecimal bitmask.

Logic Interpretation: * The value 0x1 (Hex) corresponds to 0000 0000 0001 (Binary).

This indicates that DI Channel 0 is currently sensing a High signal (Active), while all other channels (DI1 ~ DI15) are Low (Inactive).

Termination: The program runs in an infinite loop, allowing the user to trigger external sensors or switches and observe the immediate state changes. Press Ctrl + C to safely exit the process.



4.8. getdi_channel

The getdi_channel demonstration allows users to monitor the real-time status of a specific Digital Input (DI) channel. Instead of returning a full bitmask, this demo provides a simplified binary status (0 or 1) for the targeted hardware pin.

Target Channel: In this example, the program is configured to monitor DI Channel 15.

Output: Read DI 15 channel value 0x0

Logic Interpretation: The value 0x0 indicates that DI Channel 15 is currently sensing a Low signal (Inactive/Open).

If the external signal transitions to High (Active/Closed), the output will immediately switch to 0x1.

Termination: Users can observe state changes in real-time. Press Ctrl + C to stop the monitoring process, which triggers the End demo message.

```
root@icpdas:~/usb-2200/examples# ./getdi_channel
Ctrl + C to break
Read DI 15 channel value 0x0
Read DI 15 channel value 0x0
Read DI 15 channel value 0x0
Read DI 15 channel value 0x0
Read DI 15 channel value 0x0
^CEnd demo
```

4.9. di_setting

The di_functions demonstration highlights the advanced signal processing capabilities of the Digital Input (DI) channels. It covers hardware-level filtering, logic polarity adjustment, and the integrated high-speed counter functions.

1. DI Digital Filter (0.1ms unit)
Function: Sets a minimum duration that a signal must maintain to be recognized as a valid logical change.

Purpose: Eliminates high-frequency noise or contact bounce (chatter) from mechanical switches, ensuring system stability in noisy industrial environments.

2. DI Inverse Mode (Normal/Inverted)
Function: Reverses the logical interpretation of the physical input voltage.

Purpose: Matches the software logic with various sensor types (NPN/PNP) without needing to modify the application code's conditional statements.

3. DI Counter Edge (Rising/Falling)
Function: Selects which signal transition triggers the hardware counter: Rising Edge (Low to High) or Falling Edge (High to Low).

Purpose: Provides flexibility to count pulses based on specific sensor output characteristics.

4 & 5. DI Counter Reading (All / Single)
Function: Reads the accumulated pulse counts from the hardware registers.

Purpose: Ideal for flow meters, encoders, or production line counting where high-speed event tracking is required independent of software polling speed.

6 & 7. DI Counter Clear (Mask / Channel)
Function: Resets the counter values to zero, either for a group of channels (via Mask) or a specific single channel.

Purpose: Allows the system to restart counting cycles after a task is completed or during system initialization.

4.9.1. DI Digital Filter

This section demonstrates how to adjust the **Hardware Digital Filter** for all Digital Input channels. The filter ensures that an input signal must remain stable for a minimum duration before the hardware recognizes it as a valid logic change.

Operational Analysis
Current Filter Value: Initially set to 0 (Filter Disabled).

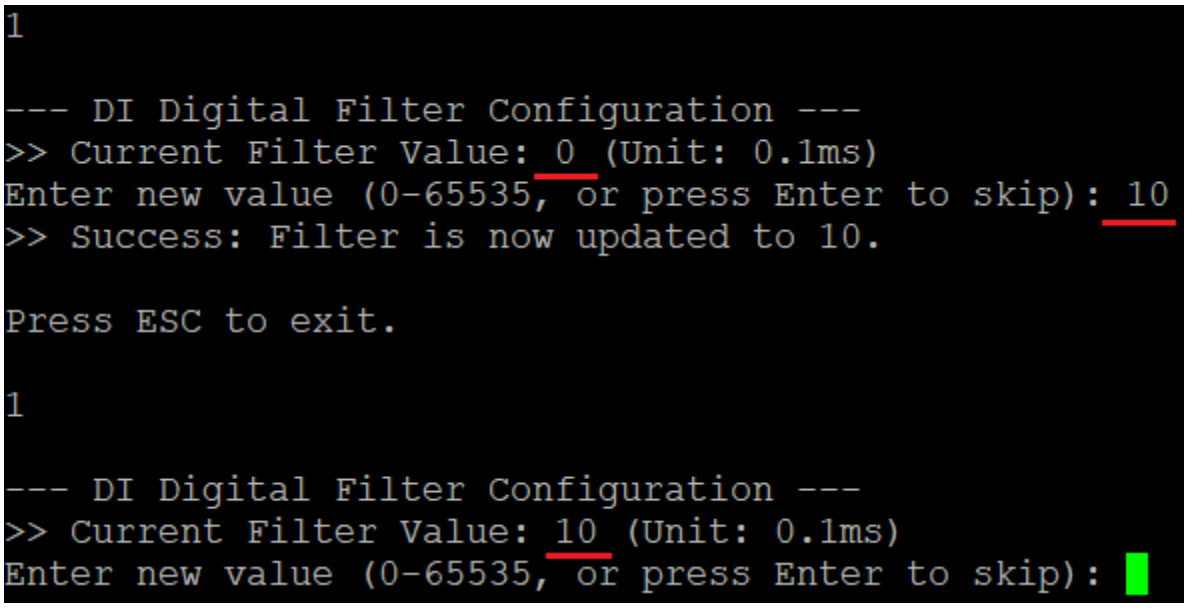
Configuration Unit: 0.1ms (100 microseconds).

Operation: Select Action (1) and enter a value (e.g., 10).

Calculation: * Input Value 10 × 0.1ms = 1.0ms.

This means any pulse or noise spike shorter than 1.0ms will be automatically ignored by the hardware.

Result: The system successfully updates the filter to 10. This provides a cleaner signal by filtering out high-frequency noise or mechanical contact chatter.



4.9.2. DI Inverse Mode

This section demonstrates how to toggle the input logic of all **Digital Input (DI)** channels between **Normal** and **Inverted** modes. This allows the software to interpret high-voltage signals as 0 and low-voltage signals as 1 (and vice versa).

Initial State: The current DI logic is [Normal].

Behavior: High Voltage (Active) = Software 1; Low Voltage (Inactive) = Software 0.

Operation: Select **Action (2)** and enter 1 to switch to **Inverted** mode.

Input: 1 (Inverted).

Result: The system updates successfully to [Inverted].

New Behavior: High Voltage (Active) = Software 0; Low Voltage (Inactive) = Software 1.

```
2
Current DI Logic: [Normal]
Enter new value (0:Normal, 1:Inverse, or press Enter to skip): 1
>> Set successfully.

Press ESC to exit.

2
Current DI Logic: [Inverted]
Enter new value (0:Normal, 1:Inverse, or press Enter to skip):
```

4.9.3. DI Counter Edge

This section demonstrates how to configure the **Edge Trigger** for the hardware counters on all Digital Input (DI) channels. The edge trigger defines the specific transition point at which the counter increments its value.

Option [0] Rising Edge (Low to High):
The counter increases when the signal voltage transitions from Low (0V) to High (24V/5V).

Use Case: Counting the start of a pulse or an "On" event.

Option [1] Falling Edge (High to Low):
The counter increases when the signal voltage transitions from High back to Low.

Use Case: Counting the end of a pulse or a "Release" event.

Scenario Trace
Initial State: Current Edge Trigger is [0] Rising Edge.

Operation: Select Action (3) and enter 1.

Result: The system updates successfully to [1] Falling Edge. From this point on, all DI counters will increment only on the falling slope of the input signal.

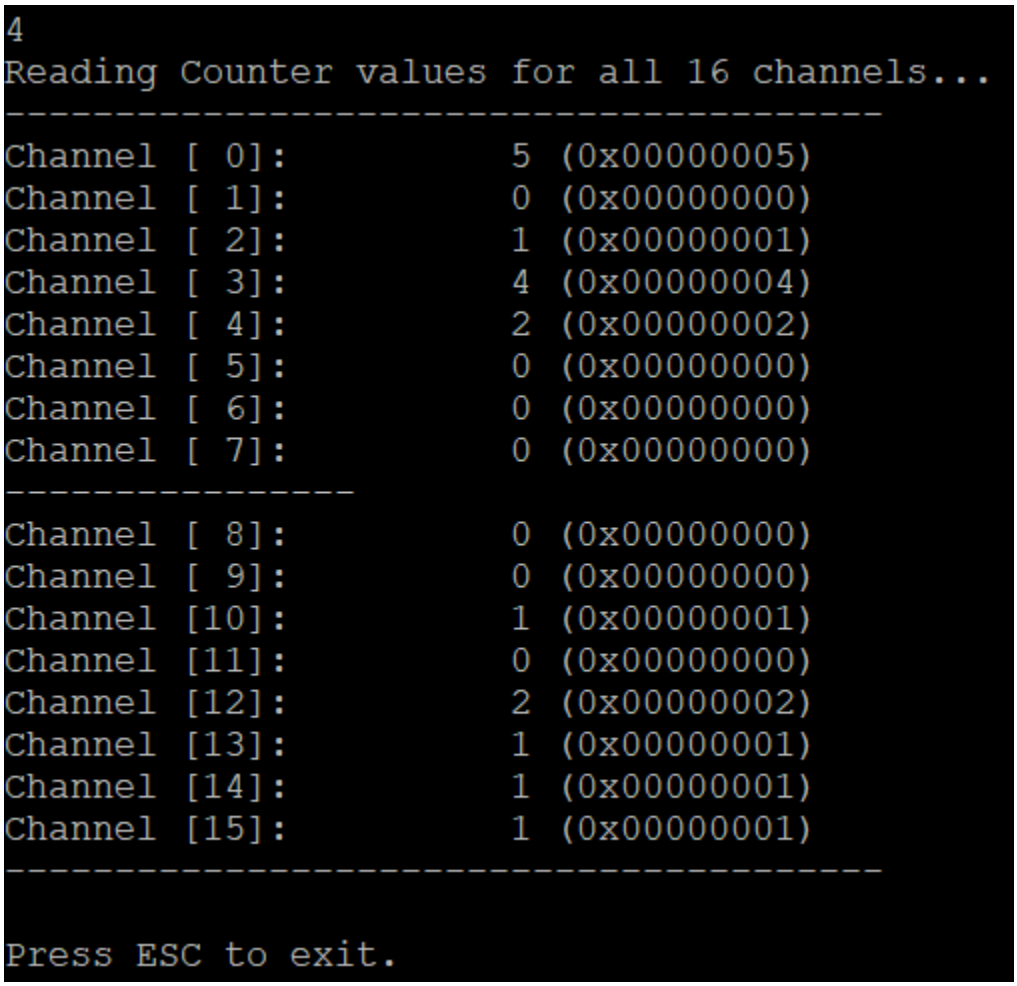
```
3

--- DI Counter Edge Trigger Configuration ---
>> Current Edge Trigger: [0] Rising Edge
Change setting? (0: Rising, 1: Falling, Enter to Skip): 1
>> Success: Edge trigger updated to [Falling]

Press ESC to exit.

3

--- DI Counter Edge Trigger Configuration ---
>> Current Edge Trigger: [1] Falling Edge
Change setting? (0: Rising, 1: Falling, Enter to Skip):
```

4.9.4. DI All Channel Counter Values

This function performs a bulk read of the internal 32-bit hardware registers for all 16 Digital Input channels. It displays the accumulated count of pulses (based on the configured Edge Trigger) for each channel in both decimal and hexadecimal formats.

Active Channels in Demo:
Channel 0: Has detected 5 pulses.
Channel 3: Has detected 4 pulses.
Channel 10-15: Have detected 1 or 2 pulses each.
Capacity: Each counter is 32-bit, meaning it can count from 0 up to 4,294,967,295 before resetting (rolling over) to zero.

Key Features
Synchronization: All 16 values are captured in a single high-speed USB transaction, ensuring that the counts across different channels are as synchronized as possible.
Background Counting: The hardware continues to count pulses even if the Linux application is busy or performing other tasks.



4.9.5. DI Single Channel Counter Value

This function allows the user to query the 32-bit hardware counter value of a specific Digital Input channel. It is particularly useful for verifying the pulse accumulation of a single sensor (e.g., a flow meter or an encoder index) without retrieving data for all channels.

Input: The user selects Action (5) and is prompted to enter a Channel Index (0-15).

Selection: In this example, Channel 0 is selected.

Output: >> [Channel 0] Counter: 5 (Hex: 0x00000005)

Decimal: The channel has registered 5 valid edge transitions.

Hexadecimal: Displayed as 0x00000005 for low-level debugging and register verification.

Accuracy: The value represents the hardware-level accumulation, which remains accurate even if the software polling frequency is low.

```
5
Read Single DI Counter. Enter Channel (0-15): 0
>> [Channel 0] Counter: 5 (Hex: 0x00000005)
```

4.9.6. DI Counter (By Mask - Multiple)

This section demonstrates how to reset multiple **Digital Input (DI) Counters** simultaneously using a **Hexadecimal Mask**. By sending a specific mask value, the hardware immediately zeros out the corresponding 32-bit registers.

Step 1: Execute Clear Command (Action 6)

Input Mask: 0xFF

Logic: 0xFF (Hex) = 0000 0000 1111 1111 (Binary).

This tells the hardware to reset DI Channel 0 through Channel 7.

Step 2: Verify Results (Action 4)

Channels [0-7]: Successfully reset to 0.

Channels [8-15]: Remained unchanged (e.g., Channel 10 still holds 1, Channel 12 still holds 2).

This proves that the mask only affects the targeted bits, allowing other counters to continue their accumulation uninterrupted.

Mask Examples for Manuals
0x01: Clear DI 0 only.

0x80: Clear DI 7 only.

0xFFFF: Clear all 16 DI channels at once.

```
6
Enter DI Counter Clear Mask (e.g., 0xFF for first 8 ch, or 0x11): 0xff
DI Counter Clear Mask [0xFF] sent successfully.

Press ESC to exit.

4
Reading Counter values for all 16 channels...
----------------------------------------
Channel [ 0]:          0 (0x00000000)
Channel [ 1]:          0 (0x00000000)
Channel [ 2]:          0 (0x00000000)
Channel [ 3]:          0 (0x00000000)
Channel [ 4]:          0 (0x00000000)
Channel [ 5]:          0 (0x00000000)
Channel [ 6]:          0 (0x00000000)
Channel [ 7]:          0 (0x00000000)
---------------
Channel [ 8]:          0 (0x00000000)
Channel [ 9]:          0 (0x00000000)
Channel [10]:          1 (0x00000001)
Channel [11]:          0 (0x00000000)
Channel [12]:          2 (0x00000002)
Channel [13]:          1 (0x00000001)
Channel [14]:          1 (0x00000001)
Channel [15]:          1 (0x00000001)
----------------------------------------
```

4.9.7. DI Counter (By Channel - Single)

This section demonstrates how to target and reset the 32-bit hardware counter of a **single Digital Input channel**. Unlike the mask-based clear, this function is designed for precision management of individual sensors.

Operational Analysis
Current State Check (Action 5):
Input: Read Channel 15.
Result: Counter: 1 (0x00000001). This confirms the channel has captured a pulse.

Execution of Clear (Action 7):
Input: Select Channel 15 for reset.
Command: USB2200_DI_ClearCounterChannel(handle, 15) is executed.
Result: Success: Counter for Channel 15 has been cleared.

Verification (Action 5):
Input: Read Channel 15 again.
Result: Counter: 0 (0x00000000). This confirms the hardware register has been successfully wiped.

```
5
Read Single DI Counter. Enter Channel (0-15): 15
>> [Channel 15] Counter: 1 (Hex: 0x00000001)

Press ESC to exit.

7

--- Clear DI Counter By Channel ---
Enter Channel ID to clear (0-15), or press Enter to cancel: 15
>> Success: Counter for Channel 15 has been cleared.

Press ESC to exit.

5
Read Single DI Counter. Enter Channel (0-15): 15
>> [Channel 15] Counter: 0 (Hex: 0x00000000)
```

Appendix

A. Error Code

This appendix provides a comprehensive reference for the error codes returned by the USB-2200 SDK functions. In industrial automation, monitoring these return values is essential for ensuring system reliability and diagnosing hardware communication issues.

Each API function returns an integer value representing the execution status. A return value of ERR_NO_ERR (typically 0) indicates that the operation was completed successfully. Any non-zero value indicates a specific error condition, such as a communication timeout, an invalid handle, or an out-of-range parameter.

Please refer to the table below to identify the cause of an error and implement the appropriate error-handling logic within your application.

Constant	Value	Description
OPEN_PARAM_ERR	-3	**Invalid Opening Parameter.** The parameters provided to the device open function (e.g., Device ID or Board ID) are incorrect.
TOO_MANY_DEVICES_ERR	-2	**Device Limit Exceeded.** The system has detected or attempted to open more USB-2200 modules than the driver supports. max devices is 32
DEV_NOTEXISTS_ERR	-1	**Device Not Found.** The specified USB-2200 module is not connected to the system or cannot be detected by the USB bus.
NO_ERR	0	**Success**
OPENDIR_ERR	1	**Directory Access Error.** The software failed to access the required system directory or driver path in the Linux environment.
DEV_NOTOPEN_ERR	2	**Device Not Opened.** The function was called before the device was successfully initialized or opened using the USB2200_OpenDevice() API.
MODULE_INIT_ERR	3	**Module Initialization Failure.** An error occurred during the hardware internal boot or firmware

		initialization process.
IDENTIFY_HEADER_ERR	4	**Protocol Header Mismatch.** The communication packet header received from the module does not match the expected USB-2200 protocol.
WRITE_BYTECOUNT_ERR	5	**Write Data Length Error.** The number of bytes successfully written to the device does not match the requested count.
READ_BYTECOUNT_ERR	6	**Read Data Length Error.** The number of bytes received from the device is inconsistent with the expected response length.
INVALID_PARAM_ERR	7	**Invalid Argument.** One or more arguments passed to the function (e.g., Channel ID, Filter Value) are out of the allowed range.
INVALID_HANDLE_ERR	8	**Invalid Device Handle.** The provided device handle is null or has been closed, making it unauthorized for current API calls.

B. Revision History

This appendix provides the revision history of the USB-2255-32/USB-2255-64 series user manual.

The table below shows the revision history.

Revision	Date	Description
1.0.0	March 2026	Initial issue